Pseudo Array Causalization

Karim Abdelhak[1] Francesco Casella[2] Bernhard Bachmann[1]

[1]Faculty of Engineering and Mathematics, University of Applied Sciences Bielefeld, Germany
{karim.abdelhak,bernhard.bachmann}@hsbi.de
[2]Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy
francesco.casella@polimi.it

Abstract

In the current state-of-the-art modeling tools for simulation, it is common to describe system behavior symbolically using mixed continuous and discrete differential-algebraic equations, so called **hybrid DAEs**. To correctly resolve higher index problems, hybrid systems and to efficiently use ODE solvers, a matching and sorting problem has to be solved, commonly referred to as **Causalization**. Typically multidimensional equations and variables are scalarized, which leads to excessive build time and generated code size in the case of large systems. In the following paper an algorithm will be presented, that preserves array structures as much as possible while still solving the problem of causalization in scalar fashion. Test results carried out in the OpenModelica tool show a reduction in build time of one/two orders of magnitude and a reduction by a factor of two/three in the simulation run time for models of the ScalableTestSuite library.

Keywords: array preservation, causalization, matching, sorting, large scale

1 Introduction

The simulation of complex physical systems typically requires the handling of large systems of hybrid differential-algebraic equations. To model such systems the object-oriented equation based language *Modelica* was developed. The development of *Modelica* drastically decreased the amount of work necessary to simulate a model based on these so called ***hybrid DAEs***. Necessary steps such as causalization, index reduction and consistent initialization have been automated using symbolic transformation.

The results of this publication have been implemented in the *OpenModelica Compiler* (Fritzson et al. 2020), which is able to compile and simulate models from different domains, such as mechanics, electrics, fluids (Braun et al. n.d.) biology (Proß and Bachmann 2011; Kofránek et al. 2010) or power systems (Casella, Leva, and Bartolini 2017; Qi 2014; Viruez et al. 2017). Generally, it is designed to simulate any model that can be described with a system of hybrid DAEs. Theoretical background and definitions for differential-algebraic equations can be found in (Mattheij and Molenaar 2002). The goal, is to create a holistic environment for modeling and simulation to be used for teaching, research and in the industry.

2 State of the Art

Current simulation tools based on the modeling language *Modelica* scalarize the equations and variables of a system to apply scalar methods of causalization and symbolic manipulations. This has major drawbacks, mainly revolving around computation time and memory usage. Besides the approach of scalarization, there has been work published with similar intentions to this paper (Otter and Elmqvist 2017; Neumayr and Otter 2023; Zimmermann, Fernández, and Kofman 2020).

The work of (Otter and Elmqvist 2017) focusses around reducing a model containing array equations to index-1 form using index-reduction methods, without having to scalarize. Although index reduction will not be covered in this paper, it is expected to be able to apply scalar methods for index reduction using pseudo array causalization.

Methods presented in (Neumayr and Otter 2023) allow the size of generated arrays to be changed after code generation. These ideas are not part of this paper, but the idea of generalized for-equations will be expanded in future work to for-equations of variable size.

The approach of (Zimmermann, Fernández, and Kofman 2020) presents a new algorithm that adapts the idea of scalarized matching and expands it to set-based graphs. Array structures are preserved as much as possible and only split up during the process of matching if no other solution can be found.

This paper focusses on providing a solution that is applicable without language restrictions while retaining the most compact array form possible. The core algorithms of matching and sorting are not expected to be bottlenecks in computation time and are performed using scalar methods, while having all other symbolic manipulation methods operate on array structures. The test results shown in section 6 support this assumption.

3 Causalization

Solving hybrid differential algebraic systems of equations requires the process of causalization, also known as BLT-Transformation[1], to ensure that

a. high differential index problems (index > 1) can be resolved

[1]BLT: Block-Lower-Triangular

b. the dependencies involving discrete equations and variables are found.

c. causalized systems can be simulated far more efficiently due to explicit assignments instead of a large implicit system. Exceptions prove the rule of course (Henningsson, Olsson, and Vanfretti 2019).

BLT-Transformation mainly consists of three steps, *Matching*, *Index-Reduction* and *Sorting*.

3.1 Scalarization

Before being able to perform *scalar* BLT-Transformation on a system of variables and equations, they need to be scalarized. **Modelica** offers comfortable ways of defining multiple equations at once, such as *for-equations*:

```
for i in 1:N loop
  der(x[i]) = i * sin(time);
end for;
```

Scalarizing this *for-equation* leads to following equations

```
der(x[1]) = 1 * sin(time);
der(x[2]) = 2 * sin(time);
...
der(x[N]) = N * sin(time);
```

which scales with the size of N. This currently is common practice among **Modelica** tools. Scalarization increases the computation time unnecessarily, as symbolic manipulation on the body is done N times instead of only once on the body equation. However, for current methods of *Causalization* this step is necessary.

Instead of scalarization, for-equations and array-equations will be converted into canonical form, which is further explained in 5.

3.2 Matching

The first step to the process of causalization is *matching*. The goal is to find an equation for each variable in which it can be solved. Note that this is only a theoretical assignment, in the sense that these assignments can also be either ambiguous, resulting in an algebraic loop, or only implicitly solvable in the first place.

A system of equations can be understood as a bipartite graph, where one set of nodes represents the equations and the other set represents the unknowns for which the system has to be solved. Edges in that graph show variable incidences in equations. The goal of *matching* algorithms is to find a *perfect matching*, which is achieved by assigning each variable uniquely to an equation such that each variable and each equation is assigned only once. This *matching* problem was analyzed thoroughly and the most commonly used algorithm to solve it is the Ford-Fulkerson (or maximum flow) algorithm, first described in (Ford and Fulkerson 1956). The OpenModelica-Compiler has a large selection of different matching algorithms, of which Pothen and Fan's algorithm (PF+) was selected to be the default (Duff, Kaya, and Uçcar 2012; Kaya et al. 2011).

3.3 Sorting

After a perfect matching has been found, the process of sorting determines the order in which to execute those assignments. The order, once again, can be ambiguous, depending on the system. Furthermore, sets of equations which have to be solved at the same time, so called *algebraic loops* are identified. Tarjan's algorithm (Tarjan 1972) is implemented in OpenModelica and the most commonly used sorting algorithm.

4 Pseudo Array Causalization

The main idea of **P**seudo **A**rray **C**ausalization (**PAC**) revolves around doing as few scalarization steps as needed, while still using scalar causalization methods. Previous tests have shown that the graph-based causalization algorithms scale linearly with the size of well posed and reasonable models, even though the theoretical computational complexity is nonlinear (Kaya et al. 2011). Far more time is spend on symbolic manipulation or generating code.

In the following an algorithm will be presented, that keeps all equations and variables in their array form only creating a scalarized graph for causalization (a different approach to (Zimmermann, Fernández, and Kofman 2020), where a set-based graph is used). Known matching (see section 3.2) and sorting (see section 3.3) algorithms can be used on the scalar graph to resolve causalization. A three-step sorting as described in section 4.3 is applied, to ensure that the result contains as few slicing steps as possible. Array-based strong components can be derived using information about the underlying array structures and the result of causalization.

Due to the array equations and variables never being scalarized symbolically, all optimization methods only scale with the number of array components rather than the number of scalarized components. Furthermore, the created simulation code is far smaller, due to the fact that compact array structures were preserved.

4.1 Preparation for Causalization

To recover the array structures after causalization with scalar methods, three structures have to be created beforehand:

Mapping: Maps array indices to the list of their scalar children and vice versa.

Matrix: Represents the scalar graph as an adjacency matrix.

Modes: Compact way of tracking which equation is solved for what variable instance.

4.1.1 Mapping

The goal of this section is to obtain functions $\mathscr{M}_V$ and $\mathscr{M}_E$ that map a variable name and its indices or an equation and its iterator values (if it is a for-equation) to a

DOI
10.3384/ecp204177

unique scalar index. Furthermore, the four inverse functions $\mathscr{M}_V^{-1}$ and $\mathscr{M}_E^{-1}$ to recover the indices or iterator values and $\hat{\mathscr{M}}_V^{-1}, \hat{\mathscr{M}}_E^{-1}$ to recover the original variables and equations from the unique scalar indices, have to be defined.

First, there needs to be a mapping for variable and equation names to their respective array indices $\mathbb{I}_a$. These are trivial, but necessary for further explanations:

$$N_V : V \to \mathbb{I}_a \tag{1}$$
$$N_E : E \to \mathbb{I}_a \tag{2}$$

with V and E as the set of variable and equation names and $\mathbb{I}_a$ being the set of array indices. Since these have to be uniquely indexed, they are bijective and have inverse functions N_V^{-1}, N_E^{-1}.

Furthermore, an index mapping for variables and equation has to be defined. The main restriction is that all scalar variables that belong to the same array variable need to have consecutive indices, the same is true for equations. This restriction allows more predictable outcomes from the causalization methods such that reasonable recollection and slicing is possible. In the following the variables will be indexed in such a way, that the innermost dimension is iterated first, then the second etc. The same is true for equations and the innermost iterator, second to innermost iterator and so on. It is important to differ between iterator *value* and *normalized index* for this indexing method. Considering an iterator with a range of $10 : -2 : 2$, the *value* 10 corresponds to *index* 0, *value* 8 corresponds to *index* 1 and so on. Even though the modeling language *Modelica* has 1-based indices, all indexing will be considered to be 0-based. This allows easier computation of index mappings. The mapping of a single variable index s_v can be done by subtracting 1. The index s_e representing an iterator value i of an equation on the other hand, has to be computed:

$$s_v(i) = i - 1 \tag{3}$$
$$s_e(i) = \frac{i - r_{start}}{r_{step}} \tag{4}$$

with r_{start} as the start and r_{step} as the step of the range. This mapping is only defined for reachable iterator values with $i \equiv r_{start} \mod r_{step}$. The function applying this index shift on all indices of a variable or iterator values of an equation will be called $S_v(indices, v)$ and $S_e(indices, e)$ respectively. For scalar variables and equations these functions return 0 if *indices* is an empty list. Furthermore these functions are bijective and therefore invertible ($\exists S_v^{-1} \wedge \exists S_e^{-1}$).

The *local* mapping of an equation or variable with n dimensions can be described as a function

$$m : \Pi_{i=1}^{n} \mathbb{I}_i \to \mathbb{I} \tag{5}$$

where $\mathbb{I}_i = \{x \in \mathbb{N}_0 \mid x < d_i\}$ and d_i being the size of the i-th dimension. $\mathbb{I} = \{x \in \mathbb{N}_0 \mid x < d\}$ with $d = \Pi_{i=1}^{n} d_i$ as the set of all flattened indices. The inverse *local* mapping

$$m^{-1} : \mathbb{I} \to \Pi_{i=1}^{n} \mathbb{I}_i \tag{6}$$

is also needed to recover the original multi-dimensional indices for slicing. Each equation and variable has their own *local* mapping m and inverse *local* mapping m^{-1}.

The two *global* mappings M_E and M_V each map all array indices to the indices of their scalar members. They are derived by creating the pseudo inverse[2] maps $M_E^\dagger$ and $M_V^\dagger$ through enumeration of all array equations ($\mathbb{I}_E^a$) and variables ($\mathbb{I}_V^a$) and all (hypothetical) scalar equations ($\mathbb{I}_E^s$) and variables ($\mathbb{I}_V^s$) and letting the scalar indices point to the index of the original array equation.

$$M_E^\dagger : \mathbb{I}_E^s \to \mathbb{I}_E^a \tag{7}$$
$$M_V^\dagger : \mathbb{I}_V^s \to \mathbb{I}_V^a \tag{8}$$

Since scalar indices have to be consecutive, the *global* mapping M can be stored more efficiently, by only storing the start index and its length. For further explanations the length is irrelevant and therefore omitted. These *global* mappings have to be created for equations ($M_E, M_E^\dagger$) and variables ($M_V, M_V^\dagger$). An example for these mappings can be found in figure 1.

In the following, variable and equation names will be used instead of their indices. Using a variable as v_{ind} with *ind* as the indices will be used as a representor of the variable index in scalar context. It is found using the following full map $\mathscr{M}_V$ which converts a variable name and its list of subscript indices to the scalar variable index.

$$v_{ind} = \mathscr{M}_V(v, ind) = M_V(N_V(v)) + m_v(S_v(ind, v)). \tag{9}$$

The inverse of this function is split up into two parts, one recovering the variable name and one recovering the indices. It will be necessary in chapter 5 to recover the original variable representations in equations.

$$v = \hat{\mathscr{M}}_V^{-1}(v_{ind}) = N_V^{-1}(M_V^\dagger(v_{ind})) \tag{10}$$
$$ind = \mathscr{M}_V^{-1}(v_{ind}) = S_v^{-1}(m_v^{-1}(v_{ind} - M_V(N_V(v))), v) \tag{11}$$

Likewise an equation as e_{val} with *val* as the iterator values implies the following operations:

$$e_{val} = \mathscr{M}_E(e, val) = M_E(N_E(e)) + m_e(S_e(val, e)). \tag{12}$$

and has similarly formulated inverse mappings:

$$e = \hat{\mathscr{M}}_E^{-1}(e_{val}) = N_E^{-1}(M_E^\dagger(e_{val})) \tag{13}$$
$$val = \mathscr{M}_E^{-1}(e_{val}) = S_e^{-1}(m_e^{-1}(e_{val} - M_E(N_E(e))), e). \tag{14}$$

[2] These pseudo inverse maps have the property $M^\dagger M = \mathrm{id}$, however in general $MM^\dagger \neq \mathrm{id}$, which is similar to the Moore-Penrose pseudo inverse matrix definition from (Penrose 1955).

```
model mapping_example
  parameter Integer n = 3;
  Real x[n+1];
  Real y[n,n];
equation
  x[1] = sin(time) "Scalar equation e";
  for i in 1:n loop
    x[i] = y[i,i] + x[i+1];
  end for "For-equation f";
  for i in 1:n, j in 1:n loop
    y[i,j] = i*cos(j*time);
  end for "For-equation g";
end mapping_example;
```

Local Mapping

m_x :
$[0] \mapsto 0$
$[1] \mapsto 1$
$[2] \mapsto 2$
$[3] \mapsto 3$

m_y :
$[0,0] \mapsto 0$
$[0,1] \mapsto 1$
$[0,2] \mapsto 2$
$[1,0] \mapsto 3$
$[1,1] \mapsto 4$
$[1,2] \mapsto 5$
$[1,3] \mapsto 6$
$[2,1] \mapsto 7$
$[2,2] \mapsto 8$

m_e :
$[0] \mapsto 0$

m_f :
$[0] \mapsto 0$
$[1] \mapsto 1$
$[2] \mapsto 2$

m_g :
$[0,0] \mapsto 0$
$[0,1] \mapsto 1$
$[0,2] \mapsto 2$
$[1,0] \mapsto 3$
$[1,1] \mapsto 4$
$[1,2] \mapsto 5$
$[2,0] \mapsto 6$
$[2,1] \mapsto 7$
$[2,2] \mapsto 8$

Global Mapping

M_V :
$0 \mapsto 0$
$1 \mapsto 4$

$M_V^\dagger$:
$0,1,2,3 \mapsto 0$
$4,5,\ldots,11,12 \mapsto 1$

M_E :
$0 \mapsto 0$
$1 \mapsto 1$
$2 \mapsto 4$

$M_E^\dagger$:
$0 \mapsto 0$
$1,2,3 \mapsto 1$
$4,5,\ldots,11,12 \mapsto 2$

Figure 1. Example for index mapping.

4.1.2 Matrix

A scalar adjacency matrix A has to be created while respecting the index mappings $\mathscr{M}_E$ and $\mathscr{M}_V$ from (7). The rows belonging to for-equations can be created by first extracting all occurring variable instances and afterwards iterating over the ranges of the iterators, replacing every instance of an iterator in the variable instance indices with their local values. For each possible iterator configuration a list of occuring variable instances is created. The multi-dimensional indices are mapped to their respective scalar index using mapping $\mathscr{M}_V$. Each iterator configuration results in the i-th row of the scalar adjacency matrix, where i is that configuration mapped using the mapping $\mathscr{M}_E$. The pseudo code for this procedure is shown in algorithm 1.

In all further explanations a bipartite graph representation of the adjacency matrix will be used where the nodes are enumerated from top to bottom. The bipartite digraph for the example from figure 1 can be seen in figure 2. The causalization modes derived from algorithm 1 are represented as edge markings and will be explained in the following section 4.1.3.

4.1.3 Modes

Each equation can be solved in n different ways, where n is the number of different variable instances in that equation. It is important to note here, that the occurence of the same variable indexed differently, has to be counted as two distinct variable instances. A causalization mode for solving an array equation e for variable v will be denominated as

$$e \twoheadrightarrow v. \tag{15}$$

Example 4.1. There are three instances `x[i]`, `y[i]` and `x[i+1]` in the following for-equation e:

```
for i in 1:10 loop
  x[i] = y[i] + x[i+1];
end for;
```

It has three causalization modes, $e \twoheadrightarrow x[i]$, $e \twoheadrightarrow y[i]$ and $e \twoheadrightarrow x[i+1]$.

For each scalar equation a mode mapping will be created as a function

$$c : \mathbb{V} \to \mathbb{M}_i \tag{16}$$

with $\mathbb{V}$ being the set of all scalar variable indices and $\mathbb{M}_i$ as the set of all modes (variable instances) for the corresponding array equation i. These mode mappings can be created alongside with the adjacency matrix A while replacing the iterators. If variable indices are used it is more efficient to create it as an inverse mapping

$$c^{-1} : \mathbb{M}_i \to \mathbb{V} \tag{17}$$

to not create large arrays when only a few variables actually occur. Since the number of modes is usually very low, one can easily traverse all modes in search for the correct scalar variable to determine in which mode the scalar equation was solved. These causalization modes can be used to correctly slice an array equation after causalization by determining the slices that have been solved for the same variable instance.

For the example from figure 1 and its digraph shown in figure 2, one can see that there are five different causalization modes in total. These causalization modes are shown in figure 3 in greater detail.

4.2 Pseudo-Array Matching

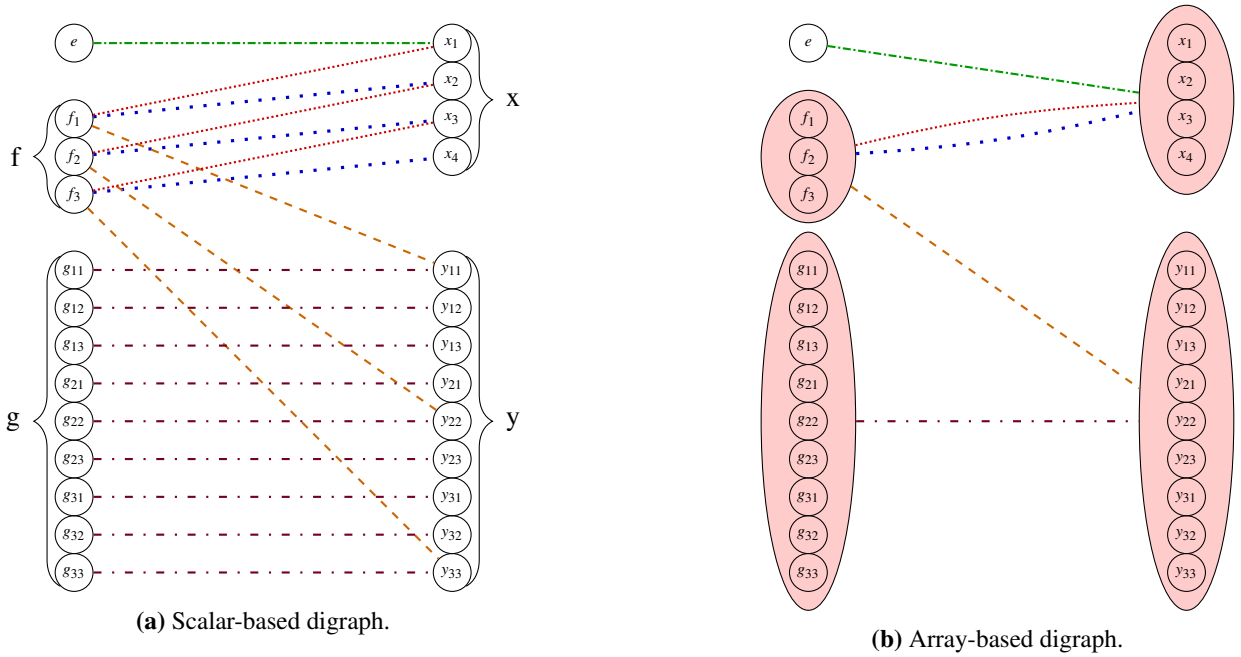
The Pseudo-Array Matching algorithm requires all the preparation described in section 4.1 and further explanations are based on the example in figure 1.

Based on the digraph shown in figure 2(a) the scalar matching algorithm described in section 3.2 is applied. The resulting perfect matching uses the causalization modes $e \twoheadrightarrow x[1], f \twoheadrightarrow x[i]$ and $g \twoheadrightarrow y[i,j]$ in its entirety. This convenient solution requires no slicing, harder problems requiring more sophisticated methods are shown in section 5. The matching solution is represented as an array Ω that maps a scalar equation index to the matched

DOI
10.3384/ecp204177

Algorithm 1 Adjacency Matrix and Causalization Modes

Input: equation array E
Input: variable array V
Output: adjacency matrix A ▷ efficient structure for matching
Output: causalization modes C (inverse map) ▷ modes to recover after matching
Output: mode to variable instance map N
$A, C, N \leftarrow$ initialize as empty arrays of size $|\mathbb{I}_E^s|$
for eq in E **do**
  $vars \leftarrow$ find all variable instances in eq using V
  $modes \leftarrow$ create unique identifiers for eq being solved for each var in $vars$
  $N[eq][modes] \leftarrow vars$ ▷ array assignment
  if eq is a for-equation **then**
    for all iterator combinations $iter$ in eq **do**
      $row \leftarrow$ apply function (9) on each var in $vars$ using $iter$
      $i \leftarrow$ apply function (12) on eq name using $iter$
      $A[i] \leftarrow$ row ▷ list assignment
      $C[i][modes] \leftarrow$ row ▷ array assignment
    end for
  else
    $row \leftarrow$ apply function (9) on each var in $vars$
    $i \leftarrow$ apply function (12) on eq
    $A[i] \leftarrow$ row ▷ list assignment
    $C[i][modes] \leftarrow$ row ▷ array assignment
  end if
end for



(a) Scalar-based digraph.

(b) Array-based digraph.

Figure 2. Digraph for model 1. The different causalization modes are as follows:
$e \rightarrow x[1]$ (green), $f \rightarrow x[i]$ (red), $f \rightarrow x[i+1]$ (blue), $f \rightarrow y[i,i]$ (orange), $g \rightarrow y[i,j]$ (purple).

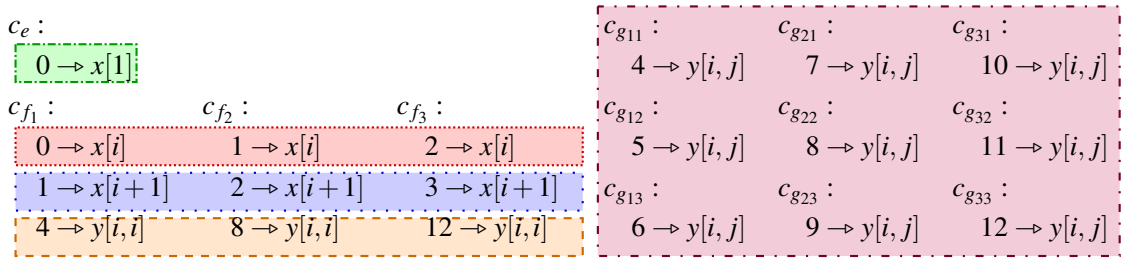


Figure 3. Scalar equation mode mapping. The colors indicate the corresponding causalization mode: $e \rightarrow x[1]$ (green), $f \rightarrow x[i]$ (red), $f \rightarrow x[i+1]$ (blue), $f \rightarrow y[i,i]$ (orange), $g \rightarrow y[i,j]$ (purple).

scalar variable index. To correctly interpret this matching in the context of array recovery, the causalization modes (see section 4.1.3) have to be taken into account. The causalization modes for the given example are shown in figure 3 but are stored as an inverse mapping for more efficient lookup by avoiding large empty arrays. The basic idea is to iterate over all modes of a given equation until the matched variable is found to determine the correct mode. A mapping, further called buckets, or in short B, that collects scalar equations that belong to the same array equation and are solved for the same variable instance (in the same causalization mode), is found by the procedure shown in algorithm 2. The bucket structure returns a list of scalar equations when provided with an array equation and a causalization mode identifier. For this trivial case, the arrays could be split up as shown in the array based digraph from figure 2(b). More complicated cases require the Three Step Sorting presented in the following chapter 4.3.

Algorithm 2 Recover Causalization Modes

```
Input: matching Ω                          ▷ eqn → var
Input: mapping M_E^-1                      ▷ scalar → array
Input: causalization modes C
Output: buckets B
B ← empty lists for all entries
for e in 0 : length(Ω) − 1 do
    m ← −1
    do
        m ← m + 1
        var ← C[e][m]
    while var ≠ Ω[e]
    append e to B(M_E^-1(e), m)
end for
```

4.3 Three Step Sorting

The result of the sorting process presented in section 3.3 does not have a unique solution and rather strongly depends on the ordering of variables and equations and even more so on the chosen mapping (see section 4.1.1). Since the result is ambiguous it can be hard to recover the most compact way of representing arrays if fragments of arrays are scattered instead of consecutive, if possible. The Three Step Sorting was implemented to ensure that the resulting sorted strong components respect the original array structures.

The three steps are *scalar sorting*, *array sorting* and *internal sorting*. A schematic outline for this process is shown in example 4.2 and the basic outline is as follows:

1. **Pseudo-Array Matching** Perform scalar matching while collecting all necessary information to recover arrays, as described in section 4.2.

2. **Scalar Sorting** The first step of sorting using Tarjan's algorithm (see section 3.3).

3. **Merge algebraic loop nodes** Merge all equation nodes that belong to the same strong component in the result of step 2 and do the same for variable nodes.

4. **Merge array nodes** Merge all equation nodes that belong to the same array and are solved for the same variable instance, using the information preserved in step 1. Do the same for variables. Equations and variables that were already merged in step 3 do not get merged in this step.

5. **Array sorting** Apply Tarjan's algorithm again on the new graph.[3]

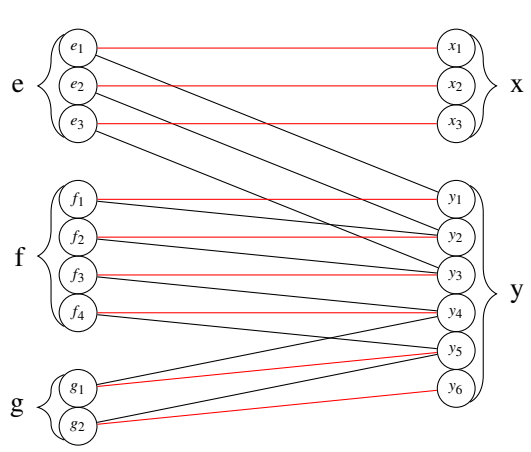
6. **Internal sorting** Strong components of size greater than one that are a result of step 5 are not algebraic loops, because these would have been found in step 2. They are equations that have to be executed sequentially, but alternate between different arrays (and/or scalar equations). These strong components will be called *entwined equations* in further explanations. Furthermore all *array* and *entwined equations* have to be sorted internally using Tarjan's algorithm.

Example 4.2 (Sliced Arrays).
As a first example for hard to solve slicing problems, we consider the following model.

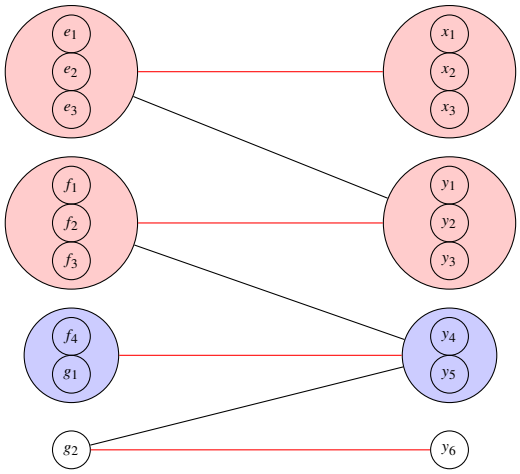
```
model sliced_arrays
  Real x[3];
  Real y[6];
```

[3] By construction each super node has scalar matching edges only to one other super node, therefore the matching for the new graph does not have to be computed.

DOI
10.3384/ecp204177



(a) Step 1: E, F, G as array equations and x, y as array variables.



(b) Step 2: Blue super nodes represent algebraic loops and red super nodes represent arrays.

Figure 4. The process of Three Step Sorting.

```
equation
  for i in 1:3 loop
    x[i] = y[i]*cos(time);
  end for "For-equation e";
  for j in 1:4 loop
    y[j] = y[j+1]*2;
  end for "For-equation f";
  for k in 5:6 loop
    y[k] = y[k-1] + sin(time);
  end for "For-equation g";
end sliced_arrays;
```

The expected outcome of this model for the process of causalization is as follows:

- The first for-equation will be matched to all of x.
- The second for-equation will be matched to index 1 to 4 of y.
- The third for-equation will be matched to index 5 and 6 of y.
- The last equation f_4 of the second for-equation forms an algebraic loop with the first equation g_1 of the third for-equation.

As can be seen in figure 4(a) the matching turns out as expected. Furthermore, one can see that there is an algebraic loop connecting the mentioned equations f_4 and g_1 and $y[4]$, $y[5]$. To efficiently resolve this loop, it is desirable to slice the for-equations in such a way, that only the two relevant equations end up in the algebraic loop and the rest is recovered as for-equations. After the first step of scalar sorting one can combine all the equations of an algebraic loop to a singular super node and do the same with variables of each algebraic loop. Only after this is done, the array super nodes should be created by merging all the remaining equations and variables to super nodes, while respecting the information gathered in section 4.1. To achieve the desired result of minimal algebraic loops while retaining as much array structure as possible, it is required to do array node merging after algebraic loop merging. The result of node merging can be seen in figure 4(b).

5 Generalized For-Equations

Before processing for-equations and array-equations they have to be converted into canonical form. Any for equation that contains $n > 1$ body equations can be split into n for-equations that each contain one single body equation. If the solution requires the body equations to be evaluated in alternating order, it will be processed as an entwined for-equation, which is explained in the following. Furthermore, array-equations can be converted into canonical for-equations using simple transformations, allthough this is not sufficiently tested yet.[4]

After the process of sorting, described in section 4.3, there are four general types of strong components.

Explicit Strong Component. An explicit strong component is a single assignment that can be solved explicitly for the chosen variable. These don't necessarily have to be scalar, they can be array assignments.

Implicit Strong Component. Implicit strong components can be single equations that cannot be solved symbolically, as well as algebraic loops, consisting of multiple equations that have to be solved simultaneously.

Simple For-Equation. A simple for-equation is a section of a for-equation that can be executed without the need to perform other assignments in between.

Entwined For-Equation. Entwined for-equations are for-equations that have mutual dependencies and

[4]Some array-equation to for-equation transformations are not efficient e.g. if they contain a function call. These will be handled as algorithms.

need to be executed in alternating order. They can also contain explicit or implicit strong components that have to be executed once.

The first two types of strong components pose no further challenge an can be processed using techniques described in section 3. The latter two for-equation based strong components require further analysis. *Simple For-Equations* have to undergo the third step of inner sorting, described in chapter 4.3, which results in a specific order of the for-equation body. This order might not be in a order that can be represented using the original for-equation iterator ranges because of slices being solved differently. An example for this is shown in the following example 5.1.

Example 5.1 (Diagonal Slice).
As an example for slices of for-equations that cannot be recovered using the original iterator range, consider a model where the diagonal of a matrix has to be solved in a different equation than the rest of it. The results for following model are shown in the two digraphs of figure 5 and confirm the expected results:

- The first for-equation will be solved for the diagonal elements of x
- The second for-equation will be split up into two for-equations:
  1. $i \neq j$ solves the remaining non-diagonal elements of x
  2. $i = j$ solves y

```
model diagonal_slice
  Real x[3,3];
  Real y[3];
equation
  for i in 1:3 loop
    x[i,i] = i*cos(time);
  end for "For-equation e";
  for i in 1:3, j in 1:3 loop
    x[i,j] = y[j] + i*sin(j*time);
  end for "For-equation f";
end diagonal_slice;
```

As can be seen in figure 5(a), the expected matching is found. No algebraic loop super nodes are created, but three different equation and variable super node pairs.

1. The first for-equation (e_1, e_2, e_3) solved for the variable instance $x[i,i]$.
2. The second for-equation $(f_{12}, f_{13}, f_{21}, f_{23}, f_{31}, f_{32})$ solved for the variable instance $x[i,j]$.
3. The second for-equation (f_{11}, f_{22}, f_{33}) solved for the variable instance $y[j]$.

The three resulting for-equations are **Simple For-Equation** strong components and need to be sorted internally. The first for-equation does not pose a problem since it is not sliced at all and has no structurally forced order. Furthermore, one can safely assume that, if nothing is forced, the sorting algorithm will act index-first and keep the equation as it is:

```
for i in 1:3 loop
  x[i,i] = i*cos(time);
end for;
```

The second for-equation poses the problem that it does not use the entirety of the original for loop. Furthermore, it cannot be represented by a single for-equation, without using an additional element, like an if-condition to strip it off its diagonal. Even though this technique could be used in this specific case, a general solution is desirable. To achieve a procedure that can be applied on for-equations sliced and ordered in any way, the list of scalar equation indices is iterated by applying algorithm 3. Trivially, the same can be done for the third for-equation.

Algorithm 3 Evaluate Generic Body

Input: Scalar index e_{val}
$e \leftarrow \hat{\mathscr{M}}_E^{-1}(e_{val})$ ▷ get equation body, see (13)
$val \leftarrow \mathscr{M}_E^{-1}(e_{val})$ ▷ get iterator values, see (14)
evaluate equation e with iterator values val

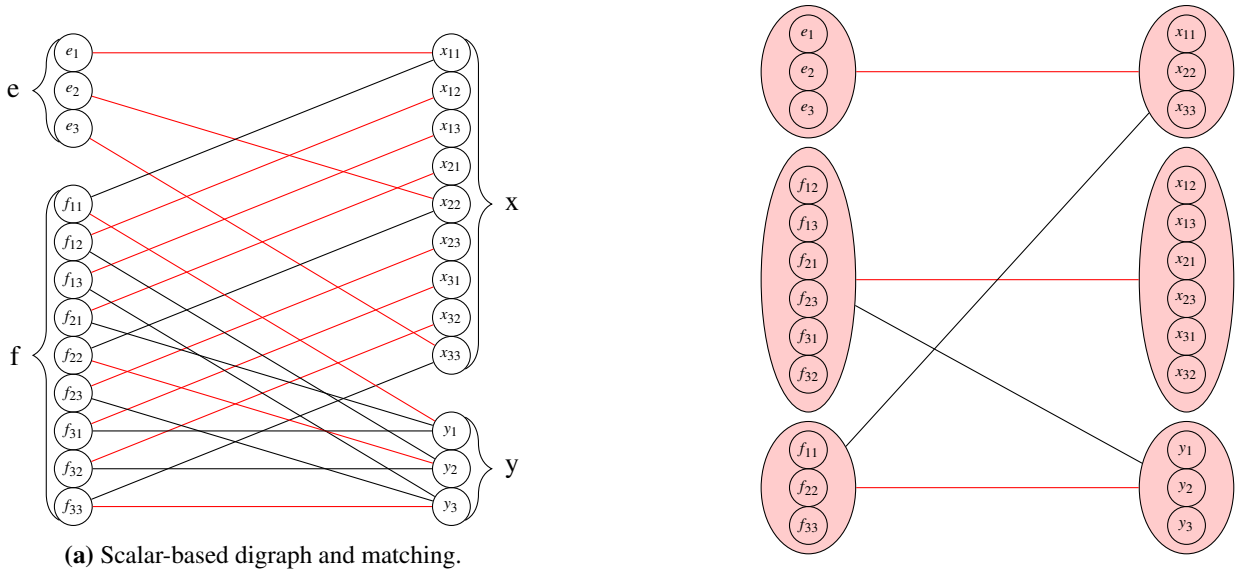
The following code is representative for the code OpenModelica generates compiling the example model `diagonal_slice`, simplified for readability. It shows the non-diagonal section of for-equation f.

```
void diagonal_slice_eq_1(DATA *data)
{
  const int idx_lst[6] = {5,2,7,1,6,3};
  for(int i=0; i<6; i++)
    genericCall_0(data, idx_lst[i]);
}
```

The `idx_lst` represents the order in which the body equations have to be solved, which is arbitrary in this specific example. This solution is provided by the sorting algorithm and no further optimization is done since this order might be enforced structurally, which is the case for other examples. The function `genericCall_0` represents the body of the function, which maps the scalar index to the iterator values, as described in algorithm 3. Allthough the theory speaks of a global mapping, in practice one uses the local mapping and the local index to evaluate the body equations of a for-equation.

```
void genericCall_0(DATA *data, int idx)
{
  int tmp = idx;
  int i_loc = tmp % 3;
  int i = 1 * i_loc + 1;
  tmp /= 3;
  int j_loc = tmp % 3;
  int j = 1 * j_loc + 1;
  tmp /= 3;
  &data->realVars[(i-1)*3+(j-1)] /*x[i,j]*/
       = &data->realVars[9+(j-1)] /*y[j]*/
      + i * sin(j * data->timeValue);
}
```


DOI
10.3384/ecp204177



(a) Scalar-based digraph and matching.

(b) Array-based digraph and matching.

Figure 5. Causalization of the example model 5.1 requiring diagonal slicing.

Example 5.2 (Entwined Loops).
As a second example for hard to solve slicing problems, we consider the following model.

```
model entwined_loops
  Real x[7];
  Real y[7];
equation
  x[1] = 1;
  y[1] = 2;
  for j in 2:7 loop
    x[j] = y[j-1] * sin(time);
  end for "For-equation e";
  for i in 2:4 loop
    y[i] = x[i-1];
  end for "For-equation f";
  for i in 5:7 loop
    y[i] = x[i-1] * 2;
  end for "For-equation g";
end entwined_loops;
```

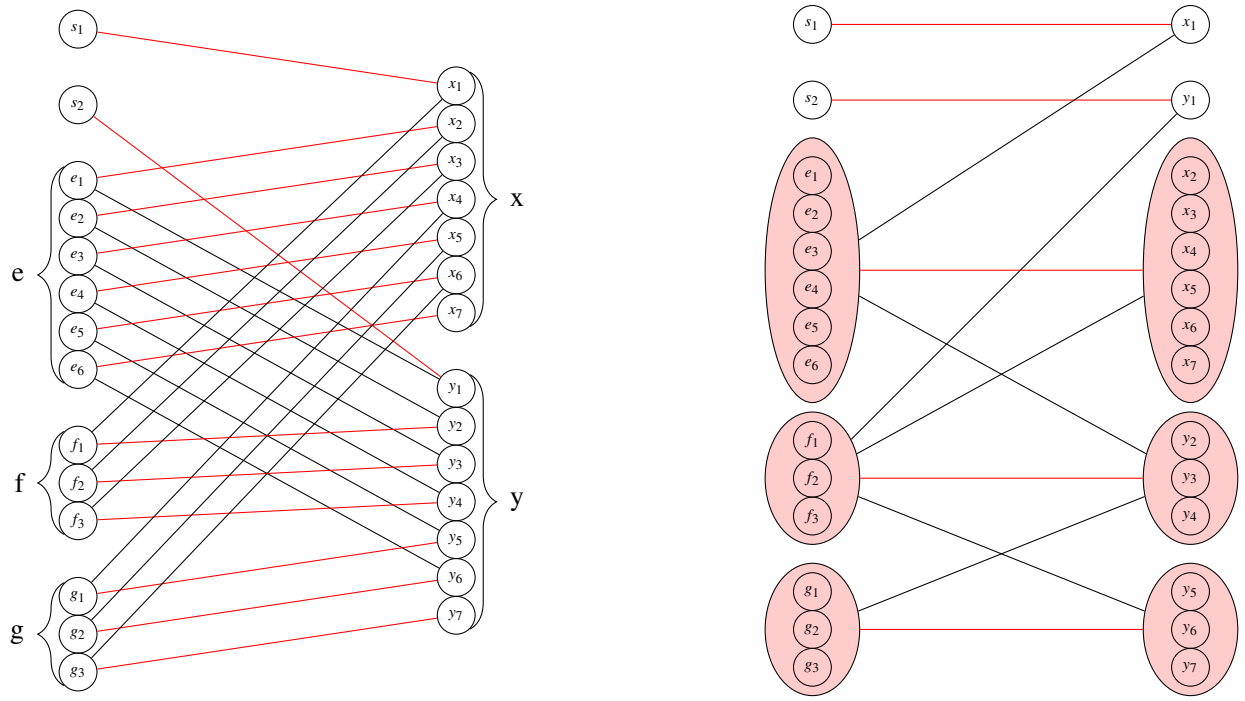
The expected results for this model are as follows:

- The first two scalar equations will be solved for $x[1]$ and $y[1]$
- The three for loops will be solved as follows:
  1. alternating between the first and the second for $i = 2:4$
  2. alternating between the first and the third for $i = 5:7$

Although this example seems more intricate, the same general solution as presented in example 5.1 can be used. The three for loops are accumulated to an entwined for-equation and its full list of alternating scalar equation indices are iterated while applying algorithm 3.

The following (simplified) code is generated by OpenModelica for the `entwined_loops` model. The body equations `genericCall_X` are similar to the one provided in example 5.1 and the alternating call order is represented by the array `call_order`.

```
void entwined_loops_eq_4(DATA *data)
{
  int call_indices[3] = {0,0,0};
  const int call_order[12] =
      {2,1,2,1,2,1,2,0,2,0,2,0};
  const int idx_lst_2[6] = {0,1,2,3,4,5};
  const int idx_lst_1[3] = {0,1,2};
  const int idx_lst_0[3] = {0,1,2};
  for(int i=0; i<12; i++)
  {
    switch(call_order[i])
    {
      case 2:
        genericCall_2(data, idx_lst_2[
            call_indices[0]]);
        call_indices[0]++;
        break;
      case 1:
        genericCall_1(data, idx_lst_1[
            call_indices[1]]);
        call_indices[1]++;
        break;
      case 0:
        genericCall_0(data, idx_lst_0[
            call_indices[2]]);
        call_indices[2]++;
        break;
      default:
        throwStreamPrint(NULL, "Call index
            %d at pos %d unknown for: ",
            call_order[i], i);
        break;
    }
  }
}
```



(a) Scalar-based digraph and matching.

(b) Array-based digraph and matching.

Figure 6. Causalization of the example model 5.2 requiring entwining.

6 Performance Test Results

The algorithms discussed in the previous Sections have been implemented in the new backend of the OpenModelica compiler. Their input is the result of the flattening process of an object-oriented Modelica model, where variable arrays and for-loop equations are *not* expanded into their scalar constituents. This output can be obtained from the new OpenModelica frontend (Pop et al. 2019), which preserves arrays during the flattening process, by skipping the final scalarization phase.

This Section reports the results obtained with large instances of some models of the ScalableTestSuite (Casella 2015), that can be run with the currently available implementation.

The tests were run on an AMD Ryzen 9 5950X 16-Core Processor, 63 GB RAM, running Ubuntu 22.04.2 LTS. The tests are run one at a time, so they can exploit parallelism on the 16 cores for garbage collection, code generation and C compilation. Also, simulations run at full speed, because they are not hindered by other processes competing for DMA channels. All simulations use variable step-size algorithms with error control.

The following models were run:

- *CascadedFirstOrder*: the model is a cascaded connection of N first-order linear systems, approximating a delay line. The response to a smooth increase of the system input is simulated with the sparse stiff solver IDA.

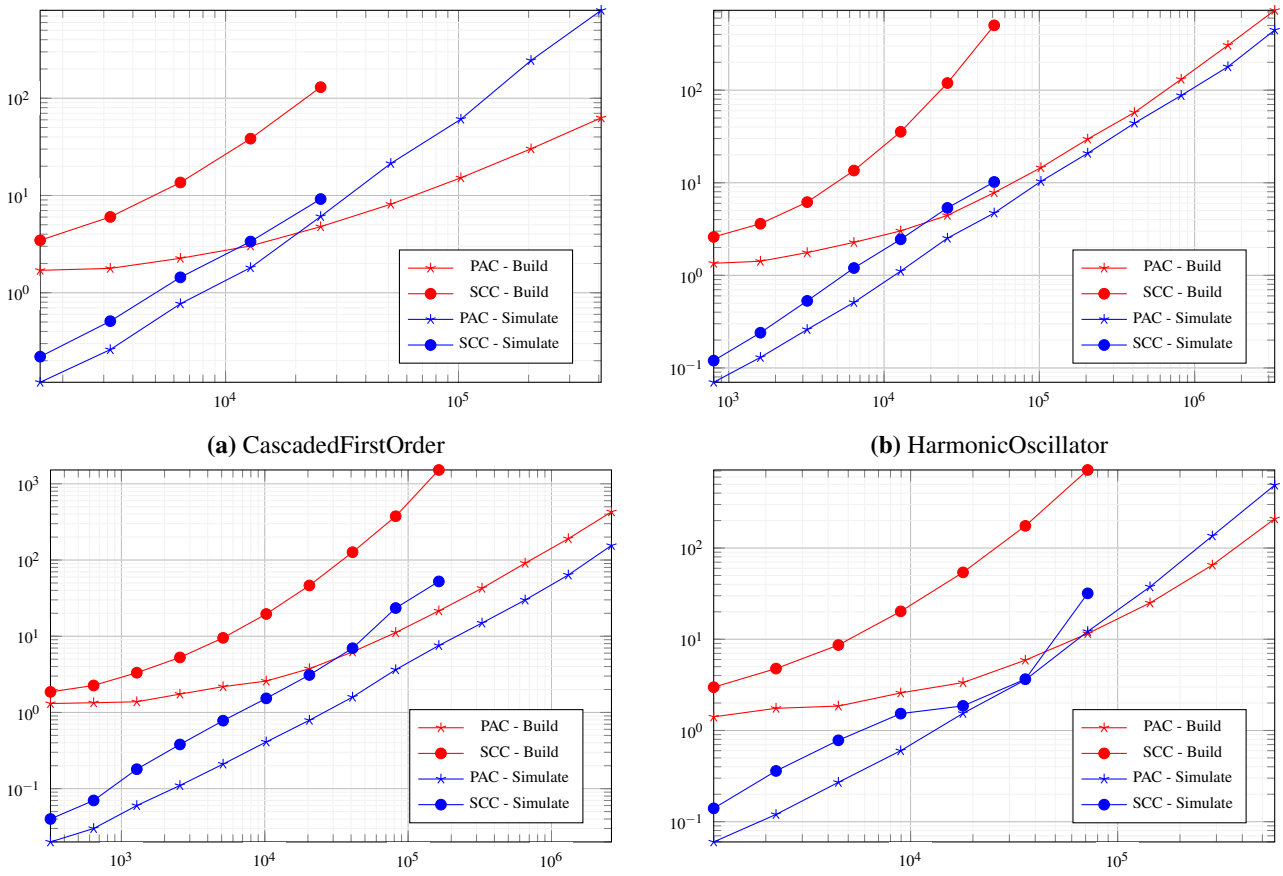
- *HarmonicOscillator*: the model describes the sequential connection of N masses with $N-1$ springs. It has $2N$ state variables and equations, where the initial position of the first mass is set off the equilibrium value, which initiates the propagation of an elastic wave through the system. As the system is only moderately stiff, the transient is simulated using the explicit DOPRI45 Runge-Kutta solver, whose execution time scales more favorably with system size.

- *OneDHeatTransferTT_FD*: this model contains the finite-volume discretization of 1D Fourier's equation, describing heat conduction in a rod, with N volumes and prescribed temperatures at the two ends. It has N states and about $2N$ equations. We simulate a transient with the sparse IDA solver, increasing the two boundary temperatures and observing how that change propagates through the length of the rod.

- *CounterCurrentHeatExchangerEquations*: this model contains a finite-volume discretization of a 1D counter-current heat exchanger model, considering the thermal inertia of the primary fluid, secondary fluid, and separating wall. Fluids are assumed to be incompressible and with constant specific heat capacity. The model has $3N$ states and $7N$ equations. We simulate a transient where we apply a step increase of the inlet temperature of one of the two fluids, using the sparse IDA solver.

Results are shown in Figure 7. On the x-axis, the number of model equations is shown; on the y-axis, build time (in red) and simulation time (in blue) are shown, compar-

DOI
10.3384/ecp204177



(a) CascadedFirstOrder

(b) HarmonicOscillator

(c) OneDHeatTransferTT_FD

(d) CounterCurrentHeatExchangerEquations

Figure 7. Comparison of Pseudo Array Causalization (PAC) to Scalarized Causalization (SCC) on a logarithmic scale. The x-axis shows the number of equations and the y-axis shows the time spent in seconds. The largest tests could not be run with SCC methods on the provided machine, due to timeout or memory overflow.

ing the pseudo-array-causalization algorithm (PAC) with the scalar causalization algorithm (SCC).

The first interesting result is that the build time with PAC is nearly constant below 1,000 equations, and only starts growing linearly for substantially large sizes, where the time spent for the scalarized causalization dominates all other phases of code generation and compilation. As a consequence, the build time is one/two orders of magnitude smaller with PAC than with SCC, with increasing advantage as the size of the model grows. In fact, PAC allows to build the code of models with size exceeding one million equations, which are simply not manageable with the SCC algorithm. The time is saved mostly by avoiding to re-run code optimizations (e.g. alias elimination or CSE) on each instance of array equation, as well as avoiding to compile huge amounts of nearly identical C-code.

The second interesting result is that simulation time is also appreciably lower with PAC, though by a constant factor of about two-three. This is probably due to the execution of for-loops in the simulation code being more efficient than the execution of similar lines of code.

Last, but not least, we observe that build time, which used to be much larger than simulation time with SCC, is now comparable or possibly shorter than simulation time. This is a key usability improvement in the model development process, which is characterized by an iterative build-simulate-analyze-modify workflow.

These results were obtained with simple models written by array variables and for-loop equations. However, large number of components of the same type can be collected into arrays, eventually leading to the same kind of array-based structure once flattened, provided that arrays are preserved during flattening.

Hence, we can claim that the PAC algorithm unlocks the possibility of handling Modelica models in the million-equations range, characterized by large arrays of variables and/or components, which was not previously practically possible with state-of-the-art Modelica tools.

7 Conclusions

In this paper, a new Pseudo-Array-Causalization (PAC) algorithm was presented. When dealing with equation-based models using arrays of variables and equations, PAC allows to first carry out the causalization process on the fully flattened bipartite graph, as it is currently done in Modelica tools, without ever scalarizing them symbolically. This allows to generate much more compact sim-

ulation code that exploits for-loops, and to do so much faster.

The presented algorithm was demonstrated in a few simple cases on the paper, but also successfully implemented and tested in the OpenModelica compiler. Early results obtained on simple but large-sized models from the ScalableTestSuite show improvements of one/two orders of magnitude in the simulation code build time, and of a factor two/three in the simulation run time, thus unlocking the possibility of simulating models with over a million equations within reasonable amounts of time.

Future work includes improving the implementation so it can be tested in realistic use cases (e.g., large transmission or distribution power system models), and most importantly handling static and dynamic index reduction.

Furthermore, generating large integer lists for the generic for-equations might become a bottle neck in the future, therefore sequence compression methods will be used to reduce the used disk space and access time.

References

Braun, Willi et al. (n.d.). "Fast Simulation of Fluid Models with Colored Jacobians". In: *Proceedings of the 9th International Modelica Conference*. DOI: 10.3384/ecp12076247. PDF.

Casella, Francesco (2015-09). "Simulation of Large-Scale Models in Modelica: State of the Art and Future Perspectives". In: *Proceedings of the 11th International Modelica Conference*, pp. 459–468. DOI: 10.3384/ecp15118459.

Casella, Francesco, Alberto Leva, and Andrea Bartolini (2017). "Simulation of Large Grids in OpenModelica: reflections and perspectives". In: *Proceedings of the 12th International Modelica Conference*.

Duff, Iain S., Kamer Kaya, and Bora Uçcar (2012). "Design, Implementation, and Analysis of Maximum Transversal Algorithms". In: *ACM Trans. Math. Softw.* 38.2. ISSN: 0098-3500. DOI: 10.1145/2049673.2049677. URL: https://doi.org/10.1145/2049673.2049677.

Ford, L. R. and D. R. Fulkerson (1956). "Maximal Flow Through a Network". In: *Canadian Journal of Mathematics* 8, pp. 399–404. DOI: 10.4153/CJM-1956-045-5.

Fritzson, Peter et al. (2020-10). "The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development". In: *Modeling, Identification and Control: A Norwegian Research Bulletin* 41, pp. 241–295. DOI: 10.4173/mic.2020.4.1.

Henningsson, Erik, Hans Olsson, and Luigi Vanfretti (2019-02). "DAE Solvers for Large-Scale Hybrid Models". In: pp. 491–502. DOI: 10.3384/ecp19157491.

Kaya, Kamer et al. (2011-01). "Experiments on Push-Relabel-based Maximum Cardinality Matching Algorithms for Bipartite Graphs". In: *Technical Report TR/PA/11/33, CER-FACS*.

Kofránek, Jiří et al. (2010). "Modelica – a language for integrative and system physiology modelling". In: *Institute of Pathophysiology, First Faculty of Medicine, Charles University, Prague*.

Mattheij, R. and J. Molenaar (2002). *Ordinary Differential Equations in Theory and Practice*. Society for Industrial and Applied Mathematics. URL: https://epubs.siam.org/doi/pdf/10.1137/1.9780898719178.

Neumayr, Andrea and Martin Otter (2023). "Modelling and Simulation of Physical Systems with Dynamically Changing Degrees of Freedom". In: *Electronics* 12.3. ISSN: 2079-9292. DOI: 10.3390/electronics12030500. URL: https://www.mdpi.com/2079-9292/12/3/500.

Otter, Martin and Hilding Elmqvist (2017). "Transformation of Differential Algebraic Array Equations to Index One Form". In: *Proceedings of the 11th International Modelica Conference*.

Penrose, R. (1955). "A generalized inverse for matrices". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 51.3, pp. 406–413. DOI: 10.1017/S0305004100030401.

Pop, Adrian et al. (2019-02). "A New OpenModelica Compiler High Performance Frontend". In: *Proceedings of the 13th International Modelica Conference*, pp. 689–698. DOI: 10.3384/ecp19157689.

Proß, Sabrina and Bernhard Bachmann (2011). "An Advanced Environment for Hybrid Modeling of Biological Systems Based on Modelica". In: *J. Integrative Bioinformatics* 8.1, pp. 1–34. DOI: 10.2390/biecoll-jib-2011-152. URL: http://dx.doi.org/10.2390/biecoll-jib-2011-152. PDF.

Qi, Le (2014). "Modelica Driven Power System: Modeling, Simulation and Validation". MA thesis. KTH Institute of Technology.

Tarjan, Robert (1972). "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2, pp. 146–160. DOI: 10.1137/0201010.

Viruez, Raul et al. (2017). "A Tool to ease Modelica-based Dynamic Power System Simulations". In: *Proceedings of the 12th International Modelica Conference*.

Zimmermann, Pablo, Joaquín Fernández, and Ernesto Kofman (2020). "Set-Based Graph Methods for Fast Equation Sorting in Large DAE Systems". In: *Proceedings of the 9th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. EOOLT '19. Berlin, Germany: Association for Computing Machinery, pp. 45–54. ISBN: 9781450377133. DOI: 10.1145/3365984.3365991. URL: https://doi.org/10.1145/3365984.3365991.